

## MIT Open Access Articles

*GAMMA: Leveraging Gustavson's Algorithm  
to Accelerate Sparse Matrix Multiplication*

The MIT Faculty has made this article openly available. **Please share**  
how this access benefits you. Your story matters.

**Citation:** Zhang, Guowei, Attaluri, Nithya, Emer, Joel and Sanchez, Daniel. 2021. "GAMMA: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication."

**As Published:** <https://doi.org/10.1145/3445814.3446702>

**Publisher:** ACM|Architectural Support for Programming Languages and Operating Systems

**Persistent URL:** <https://hdl.handle.net/1721.1/145981>

**Version:** Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

**Terms of use:** Creative Commons Attribution 4.0 International license



# GAMMA: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication

Guowei Zhang   Nithya Attaluri   Joel S. Emer   Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Cambridge, MA, USA

{zhanggw, nsattaluri, emer, sanchez}@csail.mit.edu

## ABSTRACT

Sparse matrix-sparse matrix multiplication (sPMSPM) is at the heart of a wide range of scientific and machine learning applications. sPMSPM is inefficient on general-purpose architectures, making accelerators attractive. However, prior sPMSPM accelerators use inner- or outer-product dataflows that suffer poor input or output reuse, leading to high traffic and poor performance. These prior accelerators have not explored Gustavson's algorithm, an alternative sPMSPM dataflow that does not suffer from these problems but features irregular memory access patterns that prior accelerators do not support.

We present GAMMA, an sPMSPM accelerator that uses Gustavson's algorithm to address the challenges of prior work. GAMMA performs sPMSPM's computation using specialized processing elements with simple high-radix mergers, and performs many merges in parallel to achieve high throughput. GAMMA uses a novel on-chip storage structure that combines features of both caches and explicitly managed buffers. This structure captures Gustavson's irregular reuse patterns and streams thousands of concurrent sparse fibers (i.e., lists of coordinates and values for rows or columns) with explicitly decoupled data movement. GAMMA features a new dynamic scheduling algorithm to achieve high utilization despite irregularity. We also present new preprocessing algorithms that boost GAMMA's efficiency and versatility. As a result, GAMMA outperforms prior accelerators by gmean 2.1 $\times$ , and reduces memory traffic by gmean 2.2 $\times$  and by up to 13 $\times$ .

## CCS CONCEPTS

• Computer systems organization  $\rightarrow$  Architectures.

## KEYWORDS

sparse matrix multiplication, sparse linear algebra, accelerator, Gustavson's algorithm, high-radix merge, explicit data orchestration, data movement reduction

### ACM Reference Format:

Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. GAMMA: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on*



This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '21, April 19–23, 2021, Virtual, USA

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8317-2/21/04.

<https://doi.org/10.1145/3445814.3446702>

*Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3445814.3446702>

## 1 INTRODUCTION

Scientific and machine learning applications are increasingly computing on sparse data, i.e., data where a large fraction of values are zeros. In this work, we focus on accelerating sparse matrix-sparse matrix multiplication (sPMSPM), a key kernel that lies at the heart of many sparse algorithms, like sparse deep neural networks [18, 39], sparse linear and tensor algebra [29, 57], graph analytics [16, 27], and simulation [6].

sPMSPM has two key characteristics that make it challenging to accelerate. First, sPMSPM is *bottlenecked by memory traffic and data movement*: it requires far fewer arithmetic operations per input element than dense matrix multiplication, and its inputs and outputs typically use a *compressed* representation that omits zeros but is more complicated to traverse, requiring irregular and indirect accesses. Thus, to be effective, accelerators *must minimize data movement*, rather than compute operations. Second, sPMSPM has a *rich algorithmic diversity*: it admits a wide range of *dataflows* (i.e., computation schedules) with different tradeoffs, and some dataflows have asymptotically worse performance on particular inputs. Thus, accelerators must achieve efficiency through specialization while avoiding the inefficiencies of using an inadequate sPMSPM dataflow.

Prior work has proposed sPMSPM accelerators that greatly improve performance over CPUs and GPUs. And yet, these accelerators have focused on one of two sPMSPM dataflows, *inner-product* [20, 43] or *outer-product* [37, 59], which have significant drawbacks (Sec. 2). Inner-product maximizes output reuse but sacrifices reuse of input matrices, and is inefficient with highly sparse matrices, as it is dominated by the cost of intersections that do not produce output values. By contrast, outer-product maximizes input reuse, but sacrifices output reuse, as it suffers from the cost and memory traffic of merging large partial output matrices. Prior accelerators have missed a third sPMSPM dataflow, *Gustavson's algorithm* [17],<sup>1</sup> which is often the most efficient dataflow and is widely used in CPUs and GPUs [15, 29, 52]. Gustavson's algorithm often achieves the least amount of memory traffic and requires simpler operations because it avoids the extremes of inner- and outer-product. However, Gustavson's algorithm has more *irregular reuse* across data structures, demanding a storage organization that can exploit that reuse to reduce memory traffic.

<sup>1</sup>MatRaptor [48], which was published after the submission of this work, is an accelerator that exploits Gustavson's algorithm. We discuss it briefly in Sec. 7.

To unlock the potential of sPMSPM acceleration, we propose GAMMA, the Gustavson-Algorithm Matrix-Multiplication Accelerator (Sec. 3). GAMMA combines three key features:

- (1) GAMMA uses simple processing elements (PEs) that linearly combine sparse input rows to produce each output row. PEs implement high-radix mergers that combine many input rows (e.g., 64 in our design) in a single pass, reducing work and memory accesses. Instead of expensive high-throughput mergers as in prior work [59], GAMMA uses simple scalar mergers, and relies on Gustavson’s row-level parallelism to achieve high throughput efficiently, using tens of PEs to perform many combinations in parallel. Thus, GAMMA concurrently processes *thousands* of compressed sparse *fibers*, variable-sized rows from inputs or partial outputs.
- (2) GAMMA uses a novel storage structure, FIBERCACHE, to efficiently buffer the thousands of fibers required by PEs. FIBERCACHE is organized as a cache to capture Gustavson’s irregular reuse patterns. However, FIBERCACHE is managed explicitly, like a large collection of buffers, to fetch missing fibers ahead of time and avoid PE stalls. This saves megabytes of dedicated on-chip buffers.
- (3) GAMMA dynamically schedules work across PEs to ensure high utilization and minimize memory traffic despite the irregular nature of Gustavson’s algorithm.

While Gustavson’s algorithm is an improvement over other dataflows, it still incurs excessive traffic on some inputs. To address this issue, we propose a preprocessing technique (Sec. 4), that combines row reordering and selective tiling of one matrix input. Preprocessing improves GAMMA’s performance and avoids pathologies across the full range of inputs.

We synthesize GAMMA and evaluate its performance on a wide range of sparse matrices (Sec. 6). Compared to state-of-the-art accelerators, with a similar hardware budget, GAMMA reduces total DRAM traffic by 2.2× on average, non-compulsory DRAM traffic by 12× on average, and achieves significantly higher DRAM bandwidth utilization. Moreover, GAMMA is effective on a much broader range of sparse matrices.

In summary, we make the following contributions:

- We show that prior sPMSPM accelerators have missed a key dataflow, Gustavson’s, which is often more efficient but has less regular access patterns than previously used dataflows.
- We build GAMMA, a novel sPMSPM accelerator that combines specialized PEs, a novel cache-based structure to capture Gustavson’s irregular reuse, and dynamic scheduling to achieve high utilization despite irregularity.
- We propose preprocessing techniques that boost GAMMA’s effectiveness and avoid Gustavson’s pathologies.
- We evaluate GAMMA under a broad range of matrices, showing large performance gains and memory traffic reductions over prior systems, as well as higher versatility.

## 2 BACKGROUND AND MOTIVATION

Sparse matrix-sparse matrix multiplication (sPMSPM) is widely used in deep learning inference [18, 39, 54], linear algebra [5, 29, 57], and graph analytics [16, 27] (including breadth-first search [16], maximum matching [44], cycle detection [58], triangle counting [2],

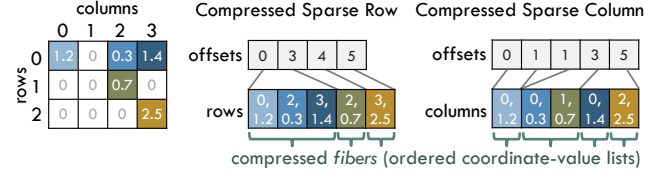


Figure 1: Compressed sparse matrix formats.

clustering [50], and all-pair shortest paths [7]). It is also a key building block for many other workloads, such as parsing [41], searching [25], and optimization [26].

We first describe the data structures used by sPMSPM and the basic sPMSPM dataflows; then, we review prior accelerators, the optimizations they introduce, and their limitations, motivating the need for a Gustavson-based accelerator.

### 2.1 Compressed Sparse Data Structures

sPMSPM operates on compressed sparse data structures, i.e., structures where only nonzeros are represented. Fig. 1 shows a sparse matrix encoded in two commonly used formats, compressed sparse row (CSR) and compressed sparse column (CSC). In CSR, rows are stored in a compressed format: each row is an ordered list of *coordinates* (in this case, column indexes) and nonzero *values*, stored contiguously. Indexing into a particular row is achieved through the offsets array, which stores the starting position of each row. CSC is analogous to CSR, but stores the matrix by compressed columns. In general, we call each compressed row or column a *fiber*, represented by a list of coordinates and values, sorted by coordinate.

Compressed sparse data structures introduce two challenges. First, certain kinds of traversals, called *concordant* traversals [49], are more efficient than others. For example, a CSR matrix can be traversed row by row, but traversing it by columns or accessing elements at random coordinates is inefficient. Thus, to be efficient, different sPMSPM dataflows impose different constraints on the preferred representation of input and output matrices. Second, sPMSPM relies on indirect accesses (through the offsets array) to variable-sized fibers, and requires combining or intersecting those fibers. These operations are inefficient on CPUs and GPUs.

### 2.2 sPMSPM Dataflows

Fig. 2 shows the three basic dataflows for sPMSPM: *inner-product*, *outer-product*, and *Gustavson*. Fig. 2 also shows the abstract loop nest corresponding to each dataflow (for simplicity, these loop nests assume dense matrices; with compressed sparse matrices, operations are more complex). sPMSPM computes  $C_{M \times N} = A_{M \times K} \times B_{K \times N}$  using a triply-nested loop that iterates over  $A$ ’s and  $B$ ’s independent dimensions,  $M$  and  $N$ , and *co-iterates* over their shared dimension,  $K$ . The dataflow is determined by the level of this co-iteration: in inner-product, co-iteration happens at the innermost loop; in outer-product, at the outermost loop; and in Gustavson’s, at the middle loop.<sup>2</sup>

<sup>2</sup>While Fig. 2 shows three loop nest orders, there are six possible orders. The remaining three stem from swapping the  $M$  and  $N$  loops; this merely switches the dimensions in which inputs are traversed, but results in an otherwise identical dataflow. For example, Fig. 2 shows an inner-product dataflow where  $A$  is traversed by rows and  $B$  by columns; swapping the outer two loops results in an inner-product dataflow where  $A$  is traversed by columns and  $B$  by rows.

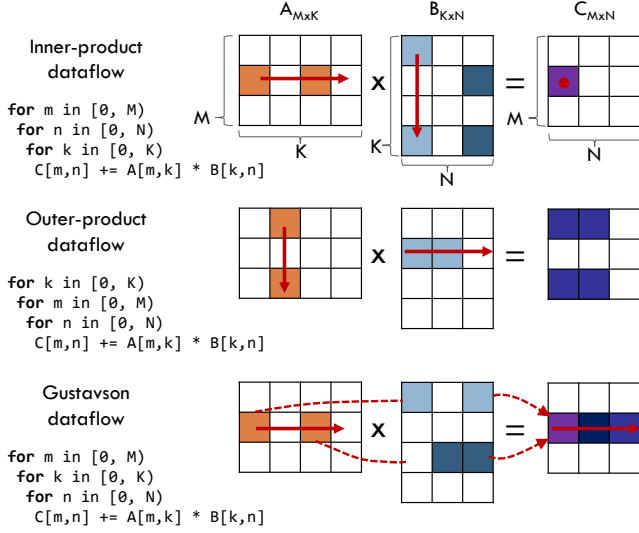


Figure 2: Comparison of basic sPMsPM dataflows.

**Inner-product** is an *output-stationary*<sup>3</sup> dataflow: it computes the output matrix one element at a time, simultaneously traversing (i.e., co-iterating) rows ( $m$ ) of  $A$  and columns ( $n$ ) of  $B$ . This achieves good output reuse, but poor input reuse. Since  $A$  and  $B$  are sparse, this traversal requires an *intersection*, as only nonzeros with matching  $k$  coordinates contribute towards the output. Inner-product is relatively efficient when the input matrices are nearly dense. But with highly sparse matrices, inner-product is dominated by the cost of intersections, which are inefficient because all elements of the rows and columns must be traversed, even though there are few *effectual* intersections, i.e., cases where both elements are nonzero. For example, in Fig. 2, intersecting row  $A_1$  and column  $B_2$  is completely ineffectual, as they have no nonzeros with the same coordinate.

**Outer-product**, by contrast, is an *input-stationary* dataflow: it computes the output one *partial matrix* at a time, traversing each column of  $A$  ( $k$ ) and row of  $B$  ( $k$ ) once and computing a full  $M \times N$  matrix that incorporates all their contributions to the output. Then, all  $K$  partial output matrices are combined to produce the final output matrix. Outer-product achieves good reuse of input matrices. Additionally, outer-product avoids inner-product's inefficiencies of ineffectual intersections: each co-iteration of a column of  $A$  and a row of  $B$  is ineffectual only when either is all-zeros, which is unlikely. However, outer-product is limited by poor output reuse: the combined size of the partial output matrices is often much larger than the final output, so they cause significant traffic. Moreover, combining these partial output matrices is a complex operation.

**Gustavson**, finally, is a *row-stationary* dataflow: it computes the output matrix one row at a time, by traversing a row of  $A$  ( $m$ ) and scaling and reducing, i.e., linearly combining, the rows of  $B$  ( $k$ ) for which the row of  $A$  has nonzero coordinates. Specifically, given a row  $A_i$  with nonzeros  $a_{ij}$ , output row  $C_i$  is produced by linearly combining  $B$ 's rows  $B_j$ , i.e.,  $C_i = \sum_j a_{ij} B_j$ . Gustavson is more efficient because it avoids the extremes of inner- and outer-product dataflows. While Gustavson does not get as much reuse of a single value as either inner- or outer-product dataflows, it gets reuse of

modestly sized rows. Unlike outer-product, Gustavson requires combining partial output *rows* rather than partial output matrices, a simpler operation on much smaller intermediates that more easily fit on-chip; and unlike inner-product, Gustavson avoids ineffectual intersections and poor input reuse.

Finally, Gustavson has an additional advantage over the other dataflows: its inputs and outputs are all in a consistent format, CSR.<sup>4</sup> By contrast, inner- or outer-product require one input to be in CSR and the other in CSC, to support efficient concordant traversals. We do not evaluate this issue further, but for compound operations (e.g., matrix exponentiation), having different formats requires expensive operand transformations, e.g., converting CSC to CSR, that rival the cost of accelerated sPMsPM [11].

### 2.3 sPMsPM Accelerators

Despite the advantages of Gustavson's algorithm, prior sPMsPM accelerators have focused on inner- and outer-product dataflows, seeking to maximize reuse of one operand. These designs incorporate different optimizations over the basic dataflow they adopt to mitigate its inefficiencies.

Accelerators like UCNN [20] and SIGMA [43] implement *inner-product* sPMsPM. These designs are built around hardware support to accelerate intersections: UCNN traverses compressed sparse data structures, while SIGMA uses a hardware-friendly bitmap-based fiber representation to further accelerate intersections. To counter poor input reuse, some designs also tile input matrices [19] to fit on-chip. While these designs achieve much higher throughput than CPUs and GPUs when matrices are relatively dense (as is typical in e.g. deep learning inference), they suffer from the algorithmic inefficiencies of ineffectual intersections on sparse matrices.

By contrast, accelerators including OuterSPACE [37], SpArch [59], and SCNN [39] implement an *outer-product* dataflow, and take different approaches to mitigate its inefficiencies. To reduce merge complexity, OuterSPACE divides partial output matrices in rows, then merges rows individually. However, OuterSPACE produces a large amount of off-chip traffic due to partial outputs, which do not fit on-chip. SpArch, by contrast, is built around a very complex high-throughput, high-radix merger that can merge up to 64 partial matrices per pass, and two main techniques to use this merger well: pipelining the production of the partial output matrices and their merging to avoid spilling them off-chip, and using a matrix condensing technique that reduces the number and size of partial output matrices. Scaling up SpArch is inefficient because its throughput is bottlenecked by the merger, and scaling up the merger's throughput incurs *quadratic* area and energy costs. Instead, GAMMA achieves high throughput with linear cost by performing many independent merges in parallel. On highly sparse matrices, SpArch often achieves nearly perfect off-chip traffic because it can produce fewer than 64 partial output matrices; however, on large or less-sparse matrices, SpArch incurs high traffic as it needs to spill many partial outputs off-chip. SpArch's matrix condensing technique also sacrifices reuse of the  $B$  matrix, which can add significant traffic.

Finally, some prior work adopts a hybrid of inner- and outer-product: ExTensor [19] is a flexible accelerator for tensor algebra that combines outer-product at the chip level, and inner-product

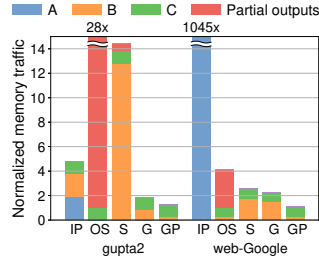
<sup>3</sup>We use the \*-stationary terminology from Chen et al. [9].

<sup>4</sup>Or CSC in the alternative Gustavson dataflow; see footnote 2.



within individual PEs. This approach requires tiling to be used well, and though this hierarchical design eliminates more ineffectual work than a pure inner-product design (by skipping entire ineffectual tiles when possible), it still suffers from the drawbacks of the dataflows it adopts.

Despite these optimizations, prior sPMSPM accelerators are saddled by the fundamental inefficiencies of the dataflows they adopt. Fig. 3 shows this by comparing the memory traffic of different accelerators when squaring (multiplying by itself) two representative sparse matrices: *gupta2* (49 MB, density  $1 \times 10^{-3}$ ), which is relatively dense, and *web-Google* (58 MB, density  $6 \times 10^{-6}$ ), which is highly sparse. We compare five accelerators with similar hardware budgets (see Sec. 5 for methodology details): (1) IP uses an inner-product dataflow with optimally tiled input matrices; (2) OS is OuterSPACE; (3) S is SpArch; (4) G is GAMMA without preprocessing; and (5) GP is GAMMA with preprocessing. Each bar shows traffic normalized to compulsory traffic (i.e., the traffic all designs would incur with unbounded on-chip memory, equivalent to reading the inputs and writing the output matrix). Traffic is broken down by data structure: reads of *A* and *B*, writes of the final output *C*, and writes and reads of partial outputs.



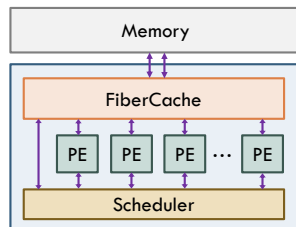
**Figure 3: Off-chip traffic of tiled inner-product (IP), OuterSPACE (OS), SpArch (S), and GAMMA without/with preprocessing (G/GP).**

Fig. 3 shows that, despite their optimizations, prior accelerators have significant drawbacks: IP works reasonably well on the denser matrix, but is inefficient on the sparser one because of many sparse tiles resulting from the hard-to-predict distribution of nonzeros. OuterSPACE suffers from partial outputs, while SpArch incurs less traffic on partial outputs, but more on matrix *B*. They both perform well on the sparser matrix, but not on the denser one. Even without preprocessing, GAMMA outperforms them all *solely by virtue of using Gustavson's dataflow*. But GAMMA supports matrix tiling and reordering techniques like prior work, as we will see in Sec. 4. With these preprocessing techniques, GAMMA achieves even larger traffic reductions. Finally, since sPMSPM is memory-bound, this lower bandwidth translates to higher performance (Sec. 6).

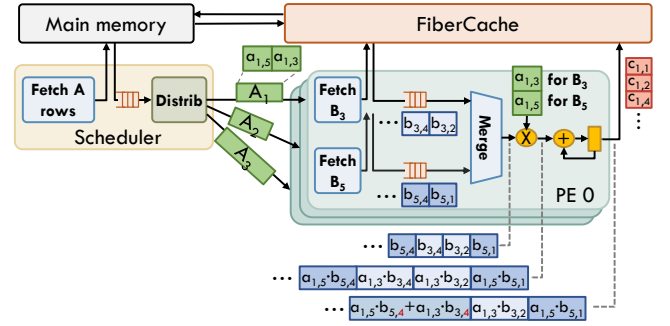
### 3 GAMMA

Fig. 4 shows an overview of GAMMA. GAMMA consists of multiple *processing elements (PEs)* that linearly combine sparse fibers; a *scheduler* that adaptively distributes work across PEs; and a *FIBERCACHE* that captures irregular reuse of fibers.

Fig. 5 illustrates GAMMA's operation through a simple example



**Figure 4: GAMMA overview.**



**Figure 5: Example showing GAMMA's operation.**

that shows how the first few elements of an output row are produced. GAMMA always operates on *fibers*, i.e., streams of nonzero values and their coordinates sorted by coordinate. First, the scheduler fetches row fibers from matrix *A* and dispatches them to PEs. Each PE then computes a *linear combination* of row fibers of *B* to produce a row fiber of output *C*. For example, in Fig. 5, the scheduler dispatches row *A*<sub>1</sub> to PE 0. Row *A*<sub>1</sub> has only two nonzeros, at coordinates 3 and 5. Therefore, PE 0 linearly combines rows *B*<sub>3</sub> and *B*<sub>5</sub>. Fig. 5 shows how the first few elements of each row are combined. First, the *B*<sub>3</sub> and *B*<sub>5</sub> fibers are streamed from the FIBERCACHE. (The FIBERCACHE retains these fibers, so subsequent uses do not incur off-chip traffic.) Then, these fibers are *merged* into a single fiber, with elements ordered by their shared (column, i.e., *N*-dimension) coordinate. Each element in the merged fiber is then scaled by the coefficient of *A*'s row corresponding to the fiber element's row (*K*) coordinate. Finally, consecutive values with the same column (*N*) coordinate are summed up, producing the output fiber. Fig. 5 shows the values of these intermediate fibers needed to produce the first three elements of output row *C*<sub>1</sub>.

**GAMMA PEs have a bounded radix, *R*:** PEs can linearly combine up to *R* input fibers in a single pass (though Fig. 5 illustrates the combination of only two fibers, GAMMA PEs have a higher radix, 64 in our implementation). When a row of *A* has more than *R* nonzeros, the scheduler breaks the linear combination into multiple rounds. For example, with *R* = 64, processing a row of *A* with 256 nonzeros would be done using four 64-way linear combinations followed by a 4-way linear combination. Each of the initial linear combinations produces a *partial output fiber*, which is then consumed by the final linear combination. The FIBERCACHE buffers these partial output fibers, avoiding off-chip traffic when possible.

**GAMMA PEs use high-radix, modest-throughput mergers:** PEs have two key design parameters: *radix*, i.e., how many input fibers they can take; and *throughput*, i.e., how many input and output elements they can consume and produce per cycle. These parameters are given by the radix and throughput of the PE's hardware *merger*, which takes *R* input fibers and produces a sequence sorted by coordinate (with repeats) as a step in creating a single output fiber from all the elements of all the input fibers. Radix and throughput choices have a substantial impact on PE and system efficiency, and on memory system design, so we discuss them first.

Implementing high-radix merges is cheap, since merger area grows linearly with radix. A high radix in turn makes computation more efficient: it allows many linear combinations to be done

in a single pass, and increasing the radix reduces the number of merge rounds and partial output fibers needed. For example, linearly combining 4096 fibers with radix-64 PEs would require 65 PE invocations in a depth-2 tree; using radix-2 PEs would require 4095 PE invocations in a depth-12 tree. The radix-64 PEs would produce one set of partial output fibers, whereas the radix-2 PEs would produce 11, increasing FIBERCACHE traffic by about an order of magnitude.<sup>5</sup>

Since higher-radix mergers are larger, there is a tradeoff between the size and power cost of the merger and both PE performance (measured in number of passes required) and FIBERCACHE traffic (due to partial output fibers). With current technology, the sweet spot balancing overall PE cost and performance occurs around  $R = 64$ .

Another consideration is the *throughput* of the merger. Implementing high-throughput mergers is costly, since merger area and energy grow *quadratically* with throughput. Producing  $N$  output elements per cycle requires the merger to consume up to  $N$  elements from a single input, and to perform up to  $N^2$  comparisons. Thus, GAMMA uses simple pipelined merge units that produce one output and consume one input per cycle, and achieves high throughput by doing many independent linear combinations in parallel, e.g., by using multiple PEs to process distinct rows of  $A$ .

This design tradeoff stands in contrast to SpArch [59], the spM-spM accelerator that comes closest to GAMMA’s efficiency. Because SpArch merges partial output *matrices* rather than fibers, it cannot exploit row-level parallelism, and implements a single high-throughput merger that dominates area and limits throughput. GAMMA and SpArch both implement radix-64 mergers. However, while in GAMMA each PE’s merger is about the same area as its floating-point multiplier, SpArch spends 38× more area on the merger than on multipliers.

**GAMMA’s on-chip storage captures irregular reuse across many fibers:** Although GAMMA’s PEs are efficient, the combination of high-radix and many PEs to achieve high throughput means that GAMMA’s memory system must support efficient accesses to a large number of concurrent fibers. For example, a system using 32 radix-64 PEs can fetch 2048 input fibers concurrently. GAMMA relies on a novel on-chip storage idiom, FIBERCACHE, to support the irregular reuse patterns of Gustavson’s algorithm efficiently. FIBERCACHE takes two key design decisions: sharing a single structure for all fibers that may have reuse, and combining caching and explicit decoupled data orchestration [40] to avoid large fetch buffers.

GAMMA processes four types of fibers: rows of A and B, and partial and final output rows of C. Rows of A and final output rows of C have no reuse, so they are streamed from/to main memory. Rows of B and partial output rows of C have reuse, but different access patterns: rows of B are read-only and are accessed potentially multiple times (depending on A's nonzeros), whereas partial output fibers, which need to be further merged to produce a final output row, are produced and consumed by PEs, typically within a short period of time. The FIBERCACHE buffers both types of fibers within a single structure, instead of having separate buffers for inputs and

<sup>5</sup>In highly sparse matrices, fibers rarely have matching coordinates, so the size of the linear combination of  $R$  fibers is close to the sum of the size of the partial output fibers (whereas for dense fibers, the final output would be a factor of  $R$  smaller).

outputs. Sharing capacity across fiber types helps because different matrices demand a widely varying share of footprint for partial outputs, but requires careful management to maximize reuse.

FIBERCACHE is organized as a highly banked cache, which allows it to flexibly share its capacity among many fibers or fiber fragments. However, FIBERCACHE is managed using the explicit data orchestration idioms common in accelerators [40]: the fibers needed by each PE are fetched ahead of time, so that when the PE reads each input fiber element, the data is served from the FIBERCACHE. This avoids PE stalls and lets the FIBERCACHE pull double duty as a latency-decoupling buffer. This feature is important because, due to the large number of concurrent fibers processed, implementing such buffering separately would be inefficient: with 32 radix-64 PEs and an 80 ns main memory, implementing these buffers would require about 2 MB of storage, a large fraction of the 3 MB FIBERCACHE we implement (Sec. 5).

### 3.1 Processing Element

Fig. 6 details the design of GAMMA’s PE. The PE linearly combines up to  $R$  fibers incrementally. Operation begins with a request from the scheduler, which streams up to  $R$  input fiber descriptors: for each input, the scheduler specifies its starting location, size, and a scaling factor. If the input fiber is a row of  $B$ ,  $B_{K_i}$ , the scaling factor is value  $a_{mk}$ ; otherwise, the input fiber is a previously generated partial output, and its scaling factor is 1.0. The PE stores scaling factors in a register file, and input fiber locations in the fiber fetcher.

The fiber fetcher then begins streaming input fibers from the FIBERCACHE. The read elements are streamed into two sets of circular buffers: coordinates ( $N$ ) are staged as inputs to the high-radix merger, while values are buffered separately. Each set has  $R$  buffers, one for each way of the merger. Since the FIBERCACHE ensures low access latency, these buffers are small and incur low overheads.

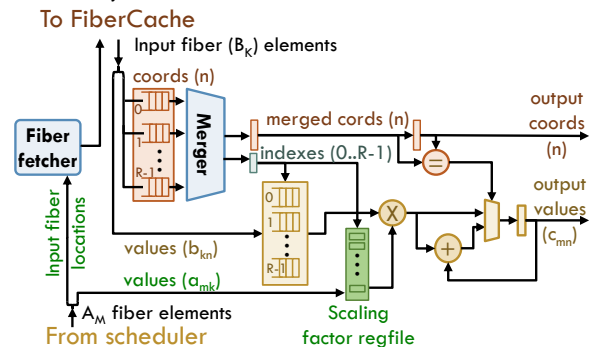


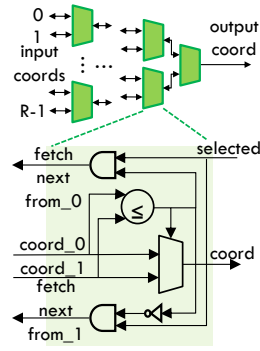
Figure 6: GAMMA’s PE architecture.

The **merger** consumes the minimum coordinate ( $N$ ) among the heads of its  $R$  input buffers, and outputs the coordinate together with its way index, i.e., a value between 0 and  $R - 1$  that identifies which input fiber this coordinate came from.

The way index is used to read both the corresponding value from the value buffer and the scaling factor. The PE then multiplies these values. Finally, the coordinate and value are processed by an accumulator that buffers and sums up the values of same-coordinate inputs. If the accumulator receives an input with a different coordinate, it emits the currently buffered element, which is part of the output fiber.

Fig. 7 shows the implementation of the merger. The merger is organized as a balanced binary tree of simple compute units. Each unit has an integer comparator for coordinates, and merges coordinate streams incrementally. This design achieves a small area cost, e.g., 55% of a 64-bit floating point multiplier for a radix of 64, and achieves an adequately high frequency.

Unlike prior mergers [45, 59] with throughputs that are high on average but are very sensitive to coordinate distribution, GAMMA's merger maintains a constant 1-element-per-cycle throughput. Thus, in steady state, the PE consumes one input fiber element per cycle and performs one scaling operation. This achieves high utilization of its most expensive components, the multiplier and the merger.



**Figure 7: High-radix merger implementation.**

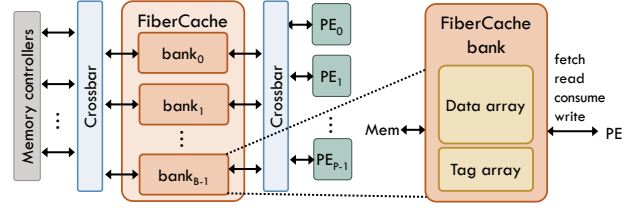
### 3.2 FIBERCACHE

Fig. 8 shows the FIBERCACHE design and interface. FIBERCACHE builds upon a cache: it has data and tag arrays, organizes data in lines, and uses a replacement policy tailored to fiber access patterns. But FIBERCACHE has two key distinct features. First, FIBERCACHE extends the usual read-write access interface with primitives that manage data movement more explicitly: *fetch* and *consume*. *fetch* enables decoupled data orchestration by fetching data from memory ahead of execution. Second, to ensure that read's hit in most cases, FIBERCACHE ensures that fetched data is unlikely to be evicted. This is achieved through the replacement policy. This effectively turns a dynamic portion of FIBERCACHE into buffer-like storage, but without incurring the high overheads of separate, statically sized buffers.

**Reading rows of  $B$**  that are not cached incurs a long latency, stalling the PE and hurting performance. FIBERCACHE addresses this issue by decoupling PE data accesses into two steps: *fetch* and *read*. A *fetch* request is sent ahead of execution and places the data into the FIBERCACHE, accessing main memory if needed, and a *read* request directs the actual data movement from FIBERCACHE to the PE. This decouples the accesses to memory and the computation on PEs.

Unlike speculative prefetching, a *fetch* is *non-speculative*: the data accessed by a *fetch* is guaranteed to have a short reuse distance. FIBERCACHE exploits this property through the replacement policy. FIBERCACHE assigns each line a priority in replacement. The priority is managed as a counter: e.g., a 5-bit counter for 32 PEs. A *fetch* request increments the priority, while a *read* request decrements it. Lower-priority lines are selected for eviction. This guarantees that most *read*'s hit in the cache; effectively, the priority is a soft lock on lines that are about to be used. FIBERCACHE uses simple 2-bit SRRIP [22] to break ties among same-priority lines.

**Reading and writing partial outputs** use the other two primitive requests: *write* and *consume*. Both *write* and *consume* exploit the fact that partial output fibers need not be backed up by memory.



**Figure 8: FIBERCACHE architecture overview.**

Upon a *write*, FIBERCACHE allocates a line without fetching it from memory, updates the data, and sets a dirty bit. A *consume* is similar to a *read*, but instead of retaining the line after the access, FIBERCACHE invalidates the line, without writing it back even though it is dirty.

**Banks and interconnect:** Since FIBERCACHE must accommodate concurrent accesses from multiple PEs, we use a highly banked design (e.g., 48 banks for 32 PEs). Banks are connected with PEs and memory controllers using crossbars.

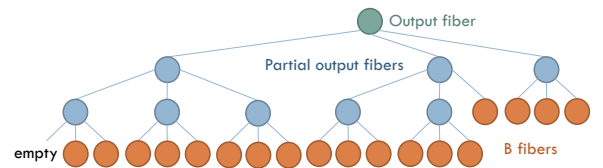
### 3.3 Scheduler

The scheduler assigns compute tasks to PEs to ensure high utilization and minimize memory traffic.

**From  $A$  to tasks:** The scheduler assigns work by traversing the rows of  $A$ . Each row of  $A$  with fewer nonzeros than the PE radix results in a single task that produces the corresponding output row and writes it directly to main memory.

When a row of  $A$  has more nonzeros  $N$  than the PE radix  $R$ , the scheduler produces a *task tree* that performs an radix- $N$  linear combination in multiple radix- $R$  steps. Fig. 9 shows an example of a task tree that combines 18 fibers using radix-3 mergers. Each node represents a fiber: the root is the output; leaves are rows of  $B$ ; and intermediate nodes are the partial output fibers. Edges denote which input fibers (children) contribute to a partial or final output fiber (parent).

The scheduler produces a *balanced, top-full tree*. *Balance* improves merge efficiency: in the common case, the rows of  $B$  have similar nonzeros, so a balanced tree results in similarly sized input fibers at each tree level. This is more efficient than a linear tree, which would build an overlong fiber. Moreover, a balanced tree enables more PEs to work on the same row in parallel. (SpArch [59] uses more sophisticated dynamic selection of merge inputs based on their lengths; this is helpful in SpArch because it purposefully constructs uneven partial output matrices, but does not help in GAMMA.) *Top-fullness* keeps footprints of partial output fibers low: by keeping the radix of the top levels full, and allowing only the lowest level to have empty input fibers, partial fibers are kept small, reducing the pressure on FIBERCACHE storage.



**Figure 9: Example schedule tree (balanced and top-full) to combine 18 input fibers on PEs with radix 3.**



**Mapping tasks to PEs:** The scheduler dynamically maps tasks to PEs: when a PE becomes ready to receive a new task, the scheduler assigns the next available one. Tasks are prioritized for execution in row order, to produce the output in an ordered fashion. For multi-task rows, the scheduler follows a dataflow (i.e., data-driven) schedule: it schedules as many leaf tasks from a single row as needed to fill PEs, and schedules each higher-level task as soon as its input fibers become available. The scheduler prioritizes higher-level tasks over lower-level ones to reduce the footprint of partial outputs.

**Staging tasks and data:** To avoid stalls when starting up a linear combination, PEs can accept a new task while processing the existing one. When a PE receives a new task, it starts staging its data into its merge buffers, so that it can switch from processing the old task to the new task in a single cycle.

The main data structure in a scheduler implementation is a scoreboard that buffers tasks not ready to dispatch and monitors partial fibers that have not been produced. Additional logic and buffers are required to fill tasks in the scoreboard by running the outermost loop of Gustavson's algorithm. The scheduler is 0.4% of total chip area.

### 3.4 Memory Management

Prior to the execution, matrices  $A$  and  $B$  are loaded into memory, and a sufficiently wide range of address space is allocated for  $C$  and partial output fibers.

Since the lengths of partial output fibers are unknown ahead of time, GAMMA allocates them dynamically. Upon scheduling a merge that produces a partial output fiber, the scheduler estimates the number of nonzeros of the fiber conservatively, by using the sum of the numbers of nonzeros in all its input fibers. The scheduler then assigns and records the address range of the partial output fiber. This space is only used if the FIBERCACHE needs to evict a partial output, a rare occurrence. The scheduler deallocates the memory when the partial output fiber is consumed. The number of partial outputs is limited to twice the number of PEs, so this dynamic memory management requires negligible on-chip memory.

## 4 PREPROCESSING FOR GAMMA

Though Gustavson is a more efficient dataflow than inner- and outer-product, it can incur high traffic. Consider Gustavson on dense operands: processing each row of  $A$  requires a complete traversal of every row of  $B$ , and results in high memory traffic. This phenomenon is mitigated for sparse operands, because processing a sparse row of  $A$  only touches a subset of rows of  $B$ , and reuse across those subsets makes the FIBERCACHE effective. Specifically, rows of  $B$  enjoy reuse in the FIBERCACHE when multiple nonzeros in  $A$  with the same column coordinate appear in nearby rows of  $A$ . However, there are two reasons this may not happen: either nearby rows of  $A$  contain largely disjoint sets of column coordinates (the matrix lacks structure), so there is minimal reuse of rows of  $B$ ; or a single row of  $A$  has many nonzeros, which requires many rows of  $B$ , thrashing the FIBERCACHE.

Prior work has addressed improving such problematic memory access patterns in sparse matrices and graphs using preprocessing techniques like tiling and reordering [21, 23, 42]. Similarly, GAMMA,

like prior accelerators, can exploit preprocessing tailored to its memory system and dataflow to further reduce data movement.

To improve data reference behavior, we design two preprocessing techniques for rows of  $A$ . *Affinity-based row-reordering* targets disparate adjacent rows of  $A$  by reordering rows so that similar rows are processed consecutively. *Selective coordinate-space tiling* breaks (only) dense rows of  $A$  into subrows to avoid thrashing, and is applied before row-reordering to extract affinity among the subrows. Both techniques can be implemented by either relying on auxiliary data for indirections or by modifying the memory layout of  $A$ . These techniques improve the reuse of sets of rows of  $B$ , achieving better versatility and efficiency.

### 4.1 Affinity-Based Row Reordering

**Problem definition:** We use a score function  $S(i, j)$  to represent the affinity of two rows  $A_i$  and  $A_j$ .  $S(i, j)$  is the number of coordinates for which both  $A_i$  and  $A_j$  have a nonzero value.

Because on-chip storage can hold rows of  $B$  corresponding to several rows of  $A$ , we are interested in maximizing the affinity of a row with the previous  $W$  adjacent rows:

$$\alpha(i) = \sum_{j=\max(0, i-W)}^{i-1} S(i, j) \quad (1)$$

We set the window size  $W$  to capture the number of rows of  $B$  that fit in the FIBERCACHE on average:

$$W = \frac{\max \text{ nnz in FIBERCACHE}}{\text{nnz per row}_A \cdot \text{nnz per row}_B} \quad (2)$$

The goal of the algorithm is to find a proper permutation of rows to maximize the affinity of the whole matrix, which we call  $\alpha$ :

$$\alpha = \sum_{i=1}^{M-1} \alpha(i) = \sum_{i=1}^{M-1} \sum_{j=\max(0, i-W)}^{i-1} S(i, j) \quad (3)$$

**Algorithm:** Algorithm 1 shows the pseudocode for the affinity-based reordering algorithm. This algorithm is greedy and uses a priority queue ( $Q$ ) to efficiently find the row with highest affinity. The algorithm produces a permutation  $P$  of  $A$ 's rows. This algorithm has complexity  $O(R \log R \cdot N^2)$ , where  $R$  is the number of rows and  $N$  is the average number of nonzeros per row, so it scales well to large matrices as long as they are sparse.

---

#### Algorithm 1: Affinity-based row reordering.

---

**Result:** Permutation  $P$  of row indices

**for**  $r \in \text{rows}$  **do**  $Q.\text{insert}(r, 0)$ ;

select some  $r$  to start,  $P[0] \leftarrow r$ ,  $Q.\text{remove}(r)$ ;

**for**  $i \in [1, M)$  **do**

**for**  $u \in \text{column coords of row } P[i-1]$  **do**

**for**  $r \in \text{row coords of column } u$  **do**

**if**  $r \in Q$  **then**  $Q.\text{incKey}(r)$ ;

**if**  $i > W$  **then**

**for**  $u \in \text{column coords of row } P[i-W-1]$  **do**

**for**  $r \in \text{row coords of column } u$  **do**

**if**  $r \in Q$  **then**  $Q.\text{decKey}(r)$ ;

$P[i] \leftarrow Q.\text{pop}()$ ;

---



## 4.2 Selective Coordinate-Space Tiling

Tiling improves input reuse (as each input tile is sized to fit on-chip) at the expense of additional intermediate outputs that must be merged. Tiling *dense* matrices is nearly always a good tradeoff [8, 38] because each input contributes to many outputs, and tiling introduces a large gain in input locality for a few extra fetches of intermediate outputs. However, this no longer holds with sparse matrices, because output traffic often dominates. In other words, tiling sparse rows may reduce traffic to  $B$  but produce many partial output fibers that must be spilled off-chip and then brought back to be merged.

Therefore, we apply tiling *selectively*, only to extremely dense rows of  $A$ . Specifically, we split rows of  $A$  whose footprint to hold rows of  $B$  is estimated to be above 25% of the FIBERCACHE capacity (the estimated footprint is the length of  $A$ 's row times the average number of nonzeros per row of  $B$ ). Each subrow resulting from this split contributes to a partial output fiber that must be combined eventually. Because these partial output fibers are *not* accessed close in time, they are likely to be spilled. To ensure that the partial output fibers generated by subrows can be combined in just one round, we use the merger's radix  $R$  as the tiling factor, i.e., the number of subrows. Rather than splitting rows into evenly-sized subrows, we perform *coordinate-space* tiling [49]: we split evenly in coordinate space, so if column coordinates are in the range  $[0, K)$ , we create up to  $R$  subrows with the  $i$ th subrow having the nonzeros within an even subrange  $[iK/R, (i+1)K/R)$ . Experimentally, we find this creates subrows with higher affinity, improving performance. In large matrices, the resulting subrows may still be large, so this process is repeated recursively.

## 5 METHODOLOGY

**System:** We evaluate a GAMMA system sized to make good use of high-bandwidth memory and consume similar levels of resources compared to prior accelerators [37, 59], in order to make fair comparisons. Our system has 32 radix-64 PEs, a 3 MB FIBERCACHE, and a 128 GB/s High-Bandwidth Memory (HBM) interface. The system runs at 1 GHz. Table 1 details the system's parameters. We built a cycle-accurate simulator to evaluate GAMMA's performance and resource utilization.

**Table 1: Configuration of the evaluated GAMMA system.**

<b>PEs</b>	32 radix-64 PEs; 1 GHz
<b>FIBERCACHE</b>	3 MB, 48 banks, 16-way set-associative
<b>Crossbars</b>	48×48 and 48×16, swizzle-switch based
<b>Main memory</b>	128 GB/s, 16 64-bit HBM channels, 8 GB/s/channel

**Table 2: Area breakdown of GAMMA (left) and one PE (right).**

	Area ( $mm^2$ )	PE component	Area ( $mm^2$ )	% PE
32 PEs	4.8	Merger	0.045	30%
Scheduler	0.11	FP Mul	0.082	55%
FIBERCACHE	22.6	FP Add	0.015	10%
Crossbars	3.1	Others	0.008	5%
Total	30.6	PE total	0.15	100%

We measure GAMMA's area by writing RTL for the PEs and scheduler. We then synthesize this logic using Synopsys Design Compiler and yosys [55] on the 45 nm FreePDK45 standard cell library [35], with a target frequency of 1GHz at 1.25V. We use CACTI 7.0 [3] to model the FIBERCACHE at 45 nm. We model the same swizzle-switch networks [46] as in prior work [37]. Table 2 shows GAMMA's area breakdown, which we contrast with prior work in Sec. 6.

**Baselines:** We compare GAMMA with two state-of-the-art accelerators, OuterSPACE and SpArch. We built detailed memory traffic models for OuterSPACE and SpArch to understand their key operational differences. We use the same approach as prior work [59] to compare end-to-end performance, by using the same set of matrices used in their evaluations. We use the original designs proposed in OuterSPACE and SpArch papers, rather than scaling them to conduct iso-area or iso-power comparisons. This is because the correct scaling strategy for each baseline is unclear. For instance, scaling SpArch requires carefully tuning various buffer and comparator array sizes. As a result, both baselines used in the comparisons have larger area than GAMMA at the same technology.

Each accelerator uses inputs in the right format for its dataflow (e.g., CSC and CSR inputs for outer-product), and SpArch uses preprocessed inputs as described by Zhang et al. [59]. We use 32-bit integer coordinates and 64-bit, double-precision floating-point

**Table 3: Characteristics of the common set of matrices (all square).**

Matrix	Nnz/row	Rows	Matrix	Nnz/row	Rows	Matrix	Nnz/row	Rows
patents_main	2.33	240,547	web-Google	5.57	916,428	2cubes_sphere	16.23	101,492
p2p-Gnutella31	2.36	62,586	scircuit	5.61	170,998	offshore	16.33	259,789
roadNet-CA	2.81	1,971,281	amazon0312	7.99	400,727	cop20k_A	21.65	121,192
webbase-1M	3.11	1,000,005	ca-CondMat	8.08	23,133	filter3D	25.43	106,437
m133-b3	4.00	200,200	email-Enron	10.02	36,692	poisson3Da	26.10	13,514
cit-Patents	4.38	3,774,768	wiki-Vote	12.50	8,297			
mario002	5.38	389,874	cage12	15.61	130,228			

**Table 4: Characteristics of the extended set of matrices.**

Matrix	Nnz/row	Rows	Cols	Matrix (Square)	Nnz/row	Rows	Matrix (Square)	Nnz/row	Rows
NotreDame_actors	3.75	392,400	127,823	gupta2	68.45	62,064	x104	80.4	108,384
relat8	3.86	345,688	12,347	vsp_bcsstk30_500	69.12	58,348	m_t1	99.96	97,578
Maragal_7	25.63	46,845	26,564	Ge87H76	69.85	112,985	ship_001	111.58	34,920
degme	43.81	185,501	659,415	raefsky3	70.22	21,200	msc10848	113.36	10,848
EternityII_Etilde	116.42	10,054	204,304	sme3Db	71.6	29,067	opt1	124.97	15,449
nemsemm1	267.17	3,945	75,352	Ge99H100	74.8	112,985	ramage02	170.31	16,830

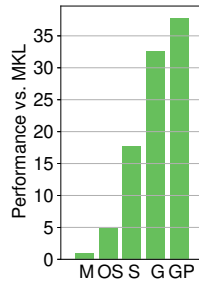
values. Because the outer-product baselines and GAMMA always consume coordinates and values, we store them together, as shown in Fig. 1. For the memory traffic comparison in Fig. 3, the inner-product accelerator (IP) uses separate coordinate and value arrays, and values are only fetched on a matching intersection, since this reduces traffic.

We also compare GAMMA against the sPMSPM implementation from Intel MKL [52] (`mk1_sparse_spm` function), running on a 4-core, 8-thread Skylake Xeon E3-1240 v5, with two DDR4-2400 channels. We do not include GPU results because existing GPU sPMSPM implementations perform similarly to MKL on CPUs [59]. **Inputs:** We use two sets of matrices. First, the **Common** set of matrices is the set used in the evaluations of OuterSPACE and SpArch, as shown in Table 3. We use the Common set for direct performance comparisons with these accelerators. However, the Common set covers only a fraction of the space of possible inputs: these matrices are square, and most are very sparse, with a maximum mean of 26 nonzeros per row. This is not representative of other commonly used matrices, and *masks the inefficiencies of outer-product designs*. To evaluate the designs with a broader range of inputs, we construct the **Extended** set of matrices, which includes 18 matrices from the SuiteSparse Matrix Collection [30]. Table 4 lists these matrices, which include non-square and square matrices with a wider range of sparsities and sizes. We evaluate  $A \times A$  for square matrices (like prior work), and  $A \times A^T$  for non-square matrices.

## 6 EVALUATION

### 6.1 Performance on Common-Set Matrices

Fig. 10 reports the performance of all accelerators on common-set matrices. Each bar shows the gmean speedup over our software baseline, MKL. Note that common-set matrices are highly sparse and thus well suited for OuterSPACE and SpArch. On these matrices, GAMMA (with preprocessing) is gmean 2.1× faster than SpArch, 7.7× faster than OuterSPACE, and 38× faster than MKL. Even without preprocessing, which makes GAMMA gmean 16% faster, GAMMA outperforms SpArch by 1.84×, OuterSPACE by 6.6×, and MKL by 33×.

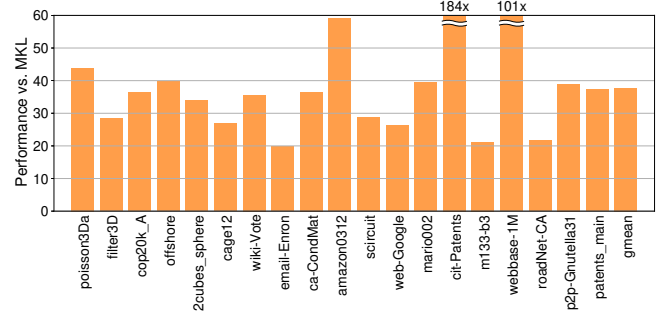


**Figure 10: Gmean speedup vs. MKL on common-set matrices for OuterSPACE (OS), SpArch (S), and GAMMA without and with preprocessing (G/GP).**

Fig. 11 further shows the per-matrix speedups of GAMMA (with preprocessing) over MKL. GAMMA outperforms MKL by up to 184×.

Fig. 12 and Fig. 13 explain how GAMMA outperforms SpArch and OuterSPACE: through a combination of reducing memory traffic and improving memory bandwidth utilization.

Fig. 12 reports the memory traffic of OuterSPACE, SpArch, and GAMMA without and with preprocessing. Each group of bars shows results for one matrix. Traffic is normalized to the compulsory traffic, which would be incurred with unbounded on-chip storage:



**Figure 11: Speedups of GAMMA with preprocessing over MKL on common-set matrices.**

fetching  $A$ , the needed rows of  $B$ , and writing  $C$ . Each bar is broken down into four categories: reads of  $A$  or  $B$ , writes of  $C$ , and reads and writes of partial outputs.

Fig. 12 shows that GAMMA incurs close-to-optimal traffic: across all inputs, it is only 7% higher than the compulsory (i.e., minimum) traffic with preprocessing, and 26% higher without preprocessing. By contrast, SpArch is 59% higher, and OuterSPACE is 4× higher. OuterSPACE suffers writes and reads to partial matrices. SpArch reduces partial output traffic over OuterSPACE, but incurs high traffic on  $B$  for two reasons. First, to reduce partial output traffic, SpArch preprocesses  $A$  to produce a schedule that worsens the access pattern to  $B$ . Second, SpArch splits its storage resources across data types (e.g., merge and prefetch buffers), leaving only part of its on-chip storage (around half a megabyte) to exploit reuse of  $B$ . By contrast, GAMMA’s shared FIBERCACHE allows  $B$ ’s rows to use more on-chip storage when beneficial. Because GAMMA’s partial outputs are rows, it has negligible partial output traffic, and its main overhead comes from imperfect reuse of  $B$ .

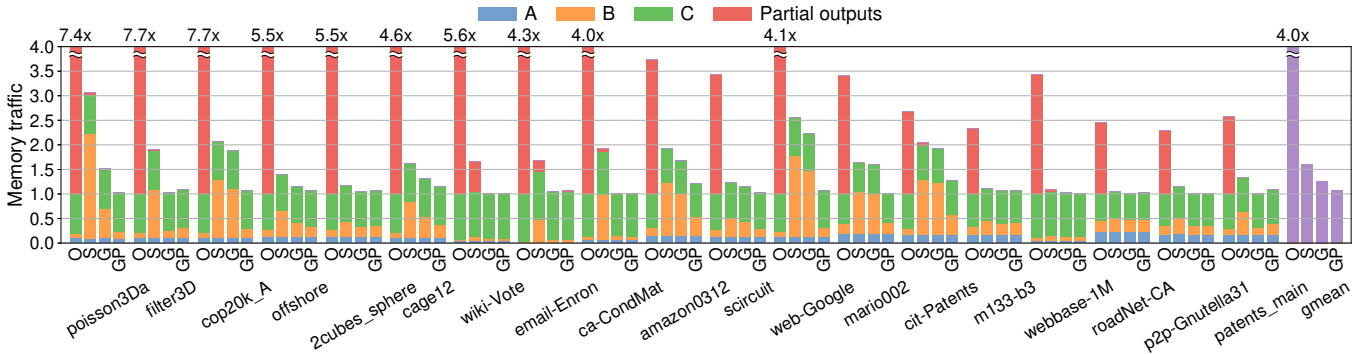
Fig. 13 further illustrates how memory bandwidth translates to performance. Because GAMMA’s PEs achieve very high throughput (processing inputs and outputs at a peak rate of 768 GB/s) and Gustavson’s algorithm does not have compute-bound execution phases, GAMMA almost always saturates the available 128 GB/s memory bandwidth. By contrast, OuterSPACE and SpArch suffer from the compute bottleneck of merging all the partial matrices, and hence achieve lower bandwidth utilizations of 48.3% and 68.6%, respectively, on the same matrices. GAMMA’s higher performance stems from its lower memory traffic and higher bandwidth utilization.

To further illustrate how FIBERCACHE is utilized, for each application, we sample the utilization of FIBERCACHE every 10,000 cycles. Fig. 14 shows the average utilization of FIBERCACHE. On these matrices,  $B$  fibers are dominant in FIBERCACHE, while partial result fibers consume non-negligible capacity on some inputs, including wiki-Vote, email-Enron, and webbase-1M.

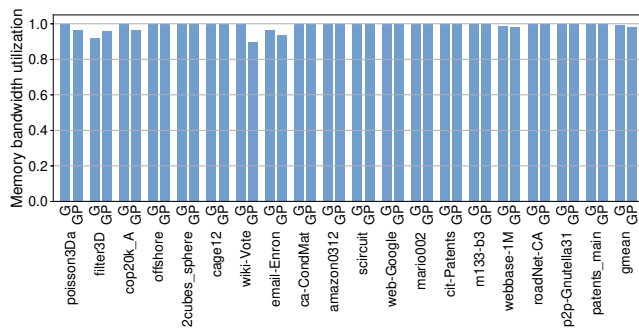
### 6.2 Performance on Extended-Set Matrices

To further evaluate the versatility of GAMMA, we use the extended set of matrices, which includes non-square matrices and square matrices more diverse than the common set (Sec. 5).

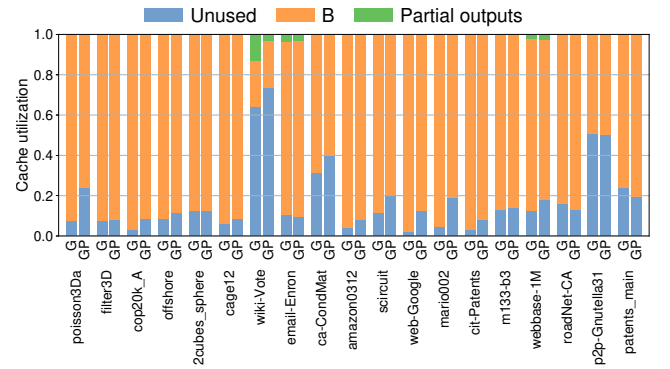
Fig. 15 shows the speedups of GAMMA (with preprocessing) over MKL. By exploiting hardware specialization, GAMMA outperforms MKL by gmean 17× and by up to 50×.



**Figure 12: Off-chip traffic on common-set matrices of OutersPACE (O), SpArch(S), and GAMMA without and with preprocessing (G/GP) (lower is better).**



**Figure 13: Memory bandwidth utilization on common-set matrices of GAMMA without and with preprocessing (G/GP).**



**Figure 14: Cache utilization on common-set matrices of GAMMA without and with preprocessing (G/GP).**

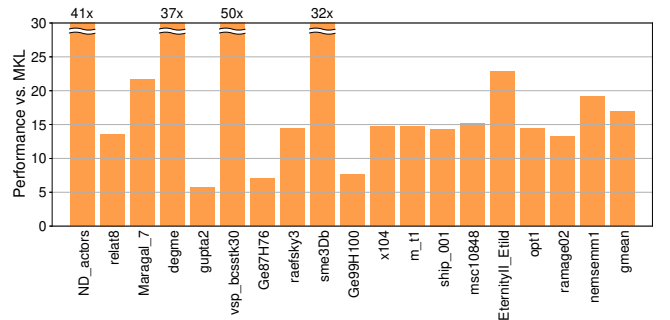
Fig. 16 compares GAMMA with SpArch and OutersPACE. The off-chip traffic of SpArch and OutersPACE are 3 $\times$  and 14 $\times$  greater than GAMMA, respectively. This difference is much larger than that in Fig. 12, because the extended set includes matrices that are denser and have more nonzeros per row. Outer-product struggles on these matrices, as it suffers from excessive memory traffic caused by writing and reading partial output matrices. For instance, on matrices that are relatively dense, such as *msc10848* and *ramage02*, such memory traffic is dominant, reaching 54 $\times$  over compulsory in OutersPACE.

Fig. 17 shows the memory bandwidth utilization of the extended-set matrices. Compared to the extremely sparse matrices in the common set, denser matrices are more bounded by compute. Therefore, some matrices in the extended set do not saturate memory bandwidth. The memory bandwidth utilization can be improved by adding more PEs to the system, as shown in Sec. 6.7.

Fig. 18 shows the utilization of FIBERCACHE on the extended-set matrices. These matrices demand a widely varying share of footprint for partial results. For instance, *ND\_actors* does not need capacity for partial results, while *Maragal\_7* spends 35% of the capacity on partial result fibers. Having a single storage structure for both *B* fibers and partial result fibers improves the versatility of the system.

### 6.3 Effectiveness of GAMMA Preprocessing

Preprocessing improves the performance of GAMMA by 18% on average. Fig. 19 further illustrates the effects of affinity-based row reordering and selective coordinate-space tiling in two cases. Affinity-based row reordering improves the reuse of *B*. For instance, it contributes to a 6 $\times$  reduction of traffic on *sme3Db*. As Sec. 4.2 explained, tiling *all* rows of *A* (+T in Fig. 19) may hurt: it does little harm to *Maragal\_7* but causes 13 $\times$  extra traffic on *sme3Db* due to excessive partial outputs. This is why GAMMA selectively tiles long rows only. Selective coordinate-space tiling reduces traffic of *B* drastically by



**Figure 15: Speedups of GAMMA with preprocessing over MKL on extended-set matrices.**

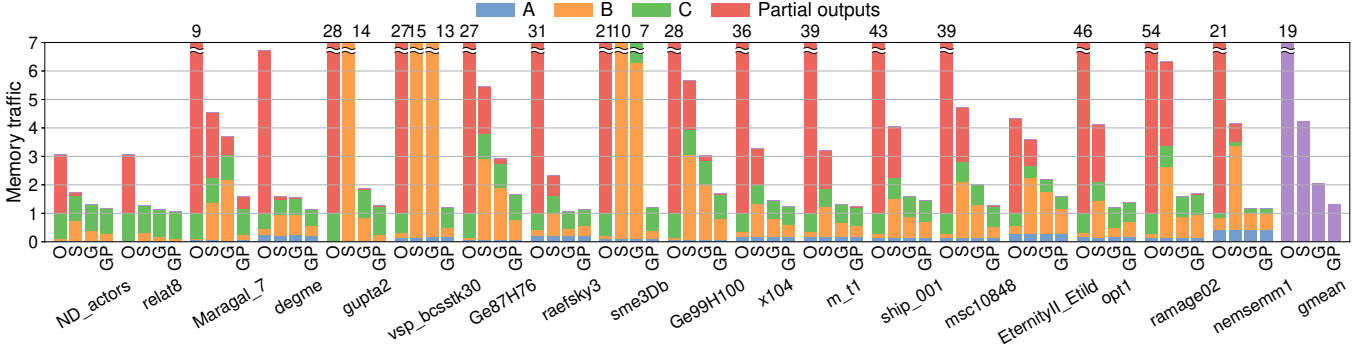


Figure 16: Off-chip traffic on extended-set matrices of OuterSPACE (O), SpArch(S), and GAMMA without and with preprocessing (G/GP) (lower is better).

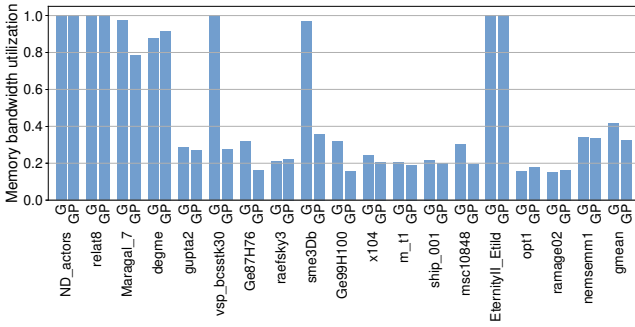


Figure 17: Memory bandwidth utilization on extended-set matrices of GAMMA without and with preprocessing (G/GP).

tiling dense rows (e.g., on Maragal\_7), and also avoids performance pathologies by not tiling sparse rows (e.g., on sme3Db).

Preprocessing takes an average time of 44 seconds and 208 seconds on the common-set matrices and the extended-set matrices, respectively. On average, the preprocessing time for a matrix is 4600× longer than using GAMMA to execute sPMsPM on the same matrix. Thus, preprocessing is beneficial only when the A matrix will be reused frequently.

#### 6.4 GAMMA Scheduling

GAMMA's scheduling algorithm (Sec. 3.3) uses multiple PEs to process the tasks produced by the same row of A (or, if preprocessing tiles the row, the same subrow of A). To demonstrate its effectiveness, we compare it against a less dynamic algorithm that always uses a single PE to process all the tasks for each row of A. Fig. 20 shows the off-chip memory traffic on input matrix email-Enron. With the single-PE approach, all the tasks from the same row are serialized, so partial result fibers stay resident in FIBERCACHE for a longer time. By contrast, GAMMA's multi-PE algorithm allows partial result fibers to be consumed as early as possible. On email-Enron, this multi-PE scheduling algorithm reduces memory traffic by 18%, and hence improves performance by 17%.

#### 6.5 GAMMA Roofline Analysis

To show that GAMMA uses resources well, Fig. 21 presents its roofline analysis plot. The plot presents arithmetic intensity (x-axis)

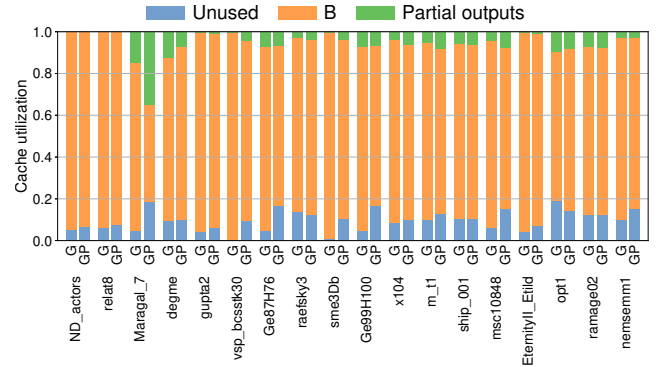


Figure 18: Cache utilization on extended-set matrices of GAMMA without and with preprocessing (G/GP).

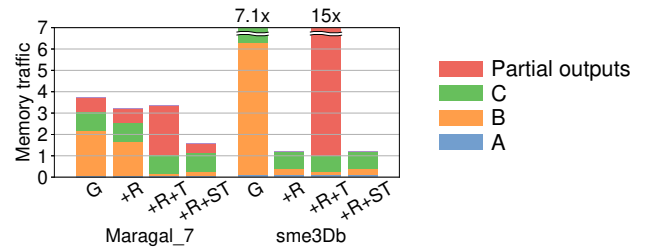


Figure 19: Off-chip traffic of GAMMA (G) and GAMMA with different preprocessing on A: affinity-based row reordering (+R), selective coordinate-space tiling (+ST), and tiling all rows (+T).

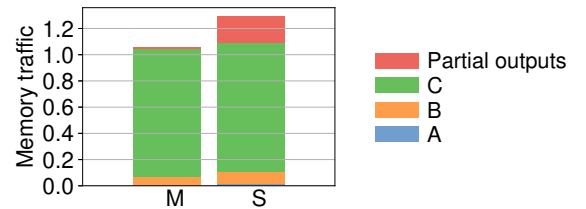
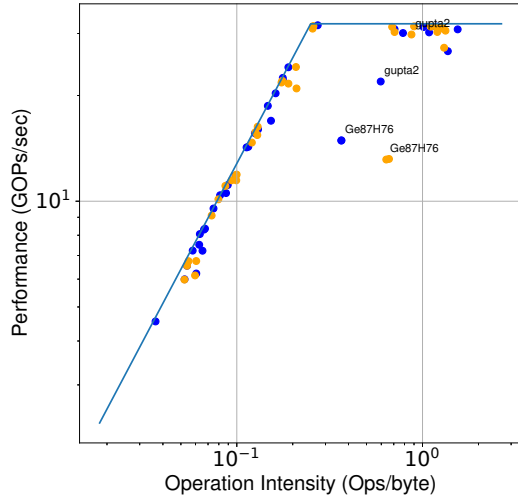


Figure 20: Off-chip traffic of GAMMA with different scheduling algorithms: using Multiple PEs (the default) or a Single PE for each row.





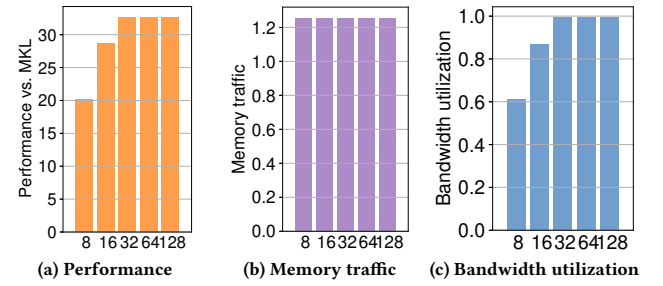
**Figure 21: Performance of GAMMA without and with preprocessing (G/CP) in a roofline model.**

in FLOPs per byte of off-chip memory traffic, and performance (y-axis) in GFLOPs (as is usual, one multiply-accumulate is counted as a single FLOP, despite being performed by a separate multiplier and adder in GAMMA PEs). Note that the plot uses a logarithmic scale for both axes. Each dot represents a single matrix; results without preprocessing are shown in blue, while results with preprocessing are shown in yellow. The plot also shows the design's roofline at 32 GFLOPs, which caps the maximum achievable performance: the sloped (left) part of the roofline is set by memory bandwidth, while the flat (right) part is set by compute (PE) throughput.

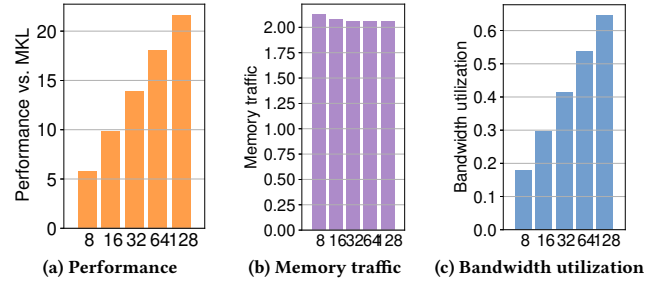
Fig. 21 shows that most matrices have low arithmetic intensity and are memory bandwidth-bound, while some have higher arithmetic intensity and are compute-bound. More importantly, this shows that GAMMA uses its resources well: almost all matrices are *right at or very close to the roofline*, showing that the system is driven to saturation all the time. Only three matrices are noticeably below the roofline (gupta2, Ge87H76, and Ge99H100). By inspection, we have found that these matrices have memory-bound and compute-bound phases, so while their average compute intensity falls past the sloped part of the roofline, they do not saturate PEs all the time due to memory-bound phases. Nonetheless, compared to prior accelerators, which have memory-bound and compute-bound phases (e.g., partial output matrix generation vs. merging in OuterSPACE and SpArch), this result shows that Gustavson's algorithm yields a more consistent behavior that uses resource better.

## 6.6 GAMMA Area Analysis

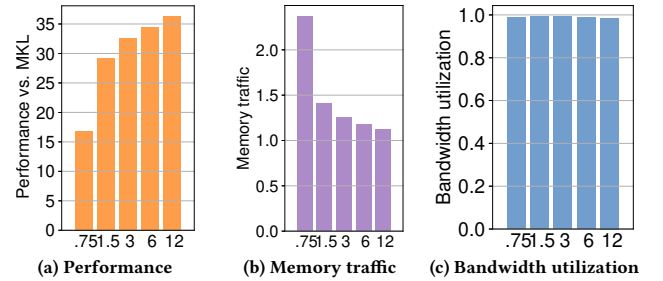
As shown in Table 2, the total area of GAMMA is  $30.6 \text{ mm}^2$ , synthesized with a 45 nm standard cell library. Scaled down to 40 nm, GAMMA's area is  $24.2 \text{ mm}^2$ , smaller than the  $28.5 \text{ mm}^2$  of SpArch at 40 nm and the  $87 \text{ mm}^2$  of OuterSPACE at 32 nm. The vast majority of area is used by the FIBERCACHE. This is a good tradeoff for SPMSpM, since the key bottleneck is memory traffic and data movement. The PEs are simple, taking 16% of chip area, and the merger and multiplier are its main components. By contrast, SpArch and OuterSPACE spend far more area on compute resources, e.g., 60% on SpArch's merger.



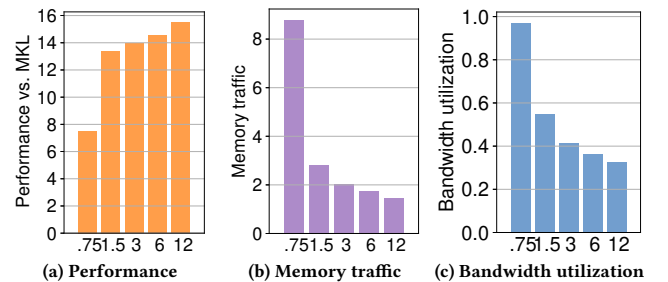
**Figure 22: Results on common-set matrices of GAMMA with different number of PEs.**



**Figure 23: Results on extended-set matrices of GAMMA with different number of PEs.**



**Figure 24: Results on common-set matrices of GAMMA with different FIBERCACHE sizes (in MB).**



**Figure 25: Results on extended-set matrices of GAMMA with different FIBERCACHE sizes (in MB).**

## 6.7 Scalability Studies

Fig. 22 and Fig. 23 show GAMMA's performance and traffic on common-set and extended-set matrices, respectively, when the number of PEs is swept from 8 to 128 (the default is 32 PEs). For common-set matrices, 32 PEs are the right tradeoff, as all are memory-bound at 32 PEs. Since some extended-set matrices have higher reuse

and thus arithmetic intensity, GAMMA continues to improve performance past 32 PEs: at 128 PEs (which would increase accelerator area by about 50%), GAMMA is gmean 65% faster than at 32 PEs.

Fig. 24 and Fig. 25 show GAMMA’s performance and traffic on common-set and extended-set matrices, respectively, when FIBER-CACHE size is swept from 0.75 MB to 12 MB (the default is 3 MB). At and after 1.5 MB, performance improves smoothly with FIBER-CACHE size, showing that GAMMA can leverage additional storage to gracefully improve performance on inputs where non-compulsory traffic is high. However, performance is significantly degraded at 0.75 MB. This performance cliff occurs because FIBER-CACHE is used as decoupling buffers, and at this size, there is little capacity left to capture irregular reuse. These results show that FIBER-CACHE does indeed save significant storage on dedicated buffers.

## 7 ADDITIONAL RELATED WORK

Much prior work has proposed optimized CPU and GPU implementations for sPMSpM, e.g., using autotuning [51], input characteristics [56], or code generation [29] to pick a well-performing sPMSpM implementation. Intel’s MKL [52], which we use in our evaluation, is generally the fastest, or close to the fastest, across input matrices [56]. Although GPUs have higher compute and memory bandwidth than CPUs, sPMSpM is a poor match to the regular data parallelism supported in current GPUs, so GPU frameworks [13, 32, 36] achieve similar sPMSpM performance to CPUs [56, 59].

Most CPU and GPU implementations follow Gustavson’s dataflow; variants differ in how they merge rows of  $B$ , e.g., using sparse accumulators [15, 28], bitmaps [24], unordered associative containers [33, 34, 36], trees [47], or heaps [1] to hold outputs. This algorithmic diversity arises because merging fibers is an expensive operation in general-purpose architectures. At a high level, heaps are space-efficient but slow, and the other data structures trade lower compute for higher space costs. GAMMA’s high-radix merges are both space-efficient and make merges very cheap, avoiding this dichotomy.

As explained in Sec. 2.3, to the best of our knowledge, accelerators earlier than GAMMA did not exploit Gustavson’s dataflow. However, MatRaptor [48], which is concurrent with GAMMA, does exploit Gustavson’s dataflow. Nonetheless, MatRaptor and GAMMA are very different. MatRaptor does not exploit the reuse of  $B$  fibers: it streams such fibers from DRAM and uses them once. By contrast, GAMMA exploits the reuse of  $B$  fibers with FIBER-CACHE. This adds area costs, but since reusing  $B$  fibers is the key way by which Gustavson’s dataflow minimizes traffic, GAMMA improves performance significantly. Consequently, on the common-set matrices, MatRaptor outperforms OuterSPACE by only 1.8× [48], worse than SpArch’s improvement over OuterSPACE (3.6×), while GAMMA outperforms OuterSPACE by 6.6× even without preprocessing.

Preprocessing of sparse matrices [10, 12, 14, 53] has been studied extensively on CPUs and GPUs. Matrix preprocessing on CPUs and GPUs typically targets creating dense tiles [42] to reduce irregularity of partial outputs, disjoint tiles [4] to minimize communication, or balanced tiles [21, 23] to ease load balancing. These techniques differ from GAMMA’s: our goal is to improve the locality of  $B$ , whereas CPUs and GPUs lack high-radix mergers and have more on-chip storage, making  $B$ ’s locality a less pressing concern.

To classify on-chip storage structures, we can use the two-dimensional taxonomy from Pellauer et al. [40]. Specifically, the content of an on-chip storage structure can be managed in two styles: *explicit* or *implicit*. Explicitly orchestrated structures allow applications to directly control what to retain or remove, while implicitly orchestrated structures infer such decisions implicitly based on read/write accesses. A storage structure can be used in either *coupled* or *decoupled* manner depending on whether the data needed is pre-staged ahead of processing to hide the memory access latency. Caches are implicit and coupled. GAMMA’s FIBER-CACHE combines features of caches and explicitly managed buffers to both exploit irregular reuse and hide memory latency through explicit decoupled data orchestration. Stash [31] is also a hybrid of caches and scratchpads, but with different goals: Stash maps data regions and accesses them explicitly, with a scratchpad interface, to reduce addressing power. Stash fetches accessed data lazily, which saves traffic when not all mapped data is accessed, but leaves accesses coupled to users. By contrast, GAMMA knows precisely which data will be accessed so its decoupled design hides long access latency. Following the taxonomy above, Stash is explicit and coupled, whereas FIBER-CACHE is implicit and decoupled.

Finally, while we focus on sPMSpM, many applications use high-dimensional tensors. For instance, TACO [28, 29] introduces workspaces and proposes compiler machinery to handle complex tensor operations. GAMMA can be combined with such techniques to support a broader range of applications.

## 8 CONCLUSION

sPMSpM is the basic building block of many emerging sparse applications, so it is crucial to accelerate it. However, prior sPMSpM accelerators use inefficient inner- and outer-product dataflows, and miss Gustavson’s more efficient dataflow. We have presented GAMMA, an sPMSpM accelerator that leverages Gustavson’s algorithm. GAMMA uses dynamically scheduled PEs with efficient high-radix mergers and performs many merges in parallel to achieve high throughput, reducing merger area by about 15× over prior work [59]. GAMMA uses a novel on-chip storage structure, FIBER-CACHE, which supports Gustavson’s irregular reuse patterns and streams thousands of concurrent sparse fibers with explicitly decoupled data movement. We also devise new preprocessing algorithms that boost GAMMA’s efficiency and versatility. As a result, GAMMA outperforms prior accelerators by gmean 2.1×, and reduces memory traffic by 2.2× on average and by up to 13×.

## ACKNOWLEDGMENTS

We sincerely thank Maleen Abeydeera, Axel Feldmann, Quan Nguyen, Nellie Yannan Wu, Nikola Samardzic, Yifan Yang, Victor Ying, Vivienne Sze; our shepherd, Christopher Hughes; and the anonymous reviewers for their helpful feedback. This work was supported in part by DARPA SDH under contract HR0011-18-3-0007 and by Semiconductor Research Corporation under contract 2020-AH-2985. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## REFERENCES

- [1] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6), 2016.
- [2] Ariful Azad, Aydin Buluc, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015.
- [3] Rajeev Balasubramanian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2), 2017.
- [4] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 3(3), 2016.
- [5] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4), 2012.
- [6] Andrew Canning, Giulia Galli, Francesco Mauri, Alessandro De Vita, and Roberto Car. O(N) tight-binding molecular dynamics on massively parallel computers: An orbital decomposition approach. *Computer Physics Communications*, 94(2-3), 1996.
- [7] Timothy M Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5), 2010.
- [8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.
- [9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd annual International Symposium on Computer Architecture (ISCA-43)*, 2016.
- [10] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009.
- [11] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [12] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, 1969.
- [13] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix-matrix multiplication for the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 41(4), 2015.
- [14] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.
- [15] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1), 1992.
- [16] John R Gilbert, Steve Reinhardt, and Viral B Shah. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, 2006.
- [17] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3), 1978.
- [18] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.
- [19] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.
- [20] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th annual International Symposium on Computer Architecture (ISCA-45)*, 2018.
- [21] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019.
- [22] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA-37)*, 2010.
- [23] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020.
- [24] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.
- [25] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the twenty-second annual symposium on Computational geometry*, 2006.
- [26] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC94)*, 1994.
- [27] Jeremy Kepner, David Bader, Aydin Buluc, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science*, 51, 2015.
- [28] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor algebra compilation with workspaces. In *Proceedings of the 17th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.
- [29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.
- [30] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The SuiteSparse matrix collection website interface. *Journal of Open Source Software*, 4(35), 2019.
- [31] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. Stash: Have your scratchpad and cache it too. In *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.
- [32] Weifeng Liu and Brian Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.
- [33] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural support for synchronization and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.
- [34] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydin Buluc. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing*, 2018.
- [35] NanGate Inc. The NanGate 45nm open cell library. [http://www.nangate.com/?page\\_id=2325](http://www.nangate.com/?page_id=2325), 2008.
- [36] Maxim Naumov, Lung-Sheng Chien, Philippe Vandermersch, and Ujval Kapasi. CUSPARSE library. In *GPU Technology Conference*, 2010.
- [37] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Apurva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*, 2018.
- [38] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Bruce Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.
- [39] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Bruce Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, 2017.
- [40] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the 24th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019.
- [41] Gerald Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1), 2006.
- [42] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC99)*, 1999.
- [43] Eric Qin, Ananda Samajdar, Hyounkjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. SIGMA: A sparse and

- irregular GEMM accelerator with flexible interconnects for dnn training. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*, 2020.
- [44] Michael O Rabin and Vijay V Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4), 1989.
- [45] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. Efficient SPMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.
- [46] Korey Sewell, Ronald G Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F Wenisch, Dennis Sylvester, et al. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2), 2012.
- [47] Sriseshan Srikanth, Anirudh Jain, Joseph M Lennon, Thomas M Conte, Erik Debenedictis, and Jeanine Cook. MetaStrider: Architectures for scalable memory-centric reduction of sparse data streams. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4), 2019.
- [48] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matpactor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of the 53rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-53)*, 2020.
- [49] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2), 2020.
- [50] Stijn Van Dongen. Performance criteria for graph clustering and Markov cluster experiments. Technical report, CWI (Centre for Mathematics and Computer Science), 2000.
- [51] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, 2005.
- [52] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi*. Springer, 2014.
- [53] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.
- [54] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016.
- [55] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2012.
- [56] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the International Conference on Supercomputing (ICS'19)*, 2019.
- [57] Ichitaro Yamazaki and Xiaoye S Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, 2010.
- [58] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the 15th annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, 2004.
- [59] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*, 2020.